bRenderer: A Flexible Basis for a Modern Computer Graphics Curriculum

Bürgisser, Benjamin ; Steiner, David ; Pajarola, Renato

Abstract: In this article, we present bRenderer, a basic educational 3D rendering framework that has resulted from four years of experience in teaching an introductory-level computer graphics course at the University of Zurich. Our renderer is based on the observation that teaching a single basic but comprehensive computer graphics course often means to face the choice between students learning a low-level graphics API bottom-up on one side, or a powerful (game) engine on the other. Solutions between these two extremes tend to be either too rudimentary to easily allow advanced visual effects in student projects, or too abstract to facilitate learning about the underlying principles of computer graphics. Our platform-independent framework abstracts the functionality of its underlying graphics API and libraries to an extent that still preserves the main concepts taught in a computer graphics course. Consequently, bRenderer can be used in student projects, as well as in exercises. It helps students to easily understand how a renderer is implemented without getting distracted by the particular implementation of the framework or platform-specific characteristics.

DOI: https://doi.org/10.2312/eged.20171023

bRenderer: A Flexible Basis for a Modern Computer Graphics Curriculum

B. Bürgisser, D. Steiner, and R. Pajarola

Visualization and MultiMedia Lab,
Department of Informatics, University of Zurich

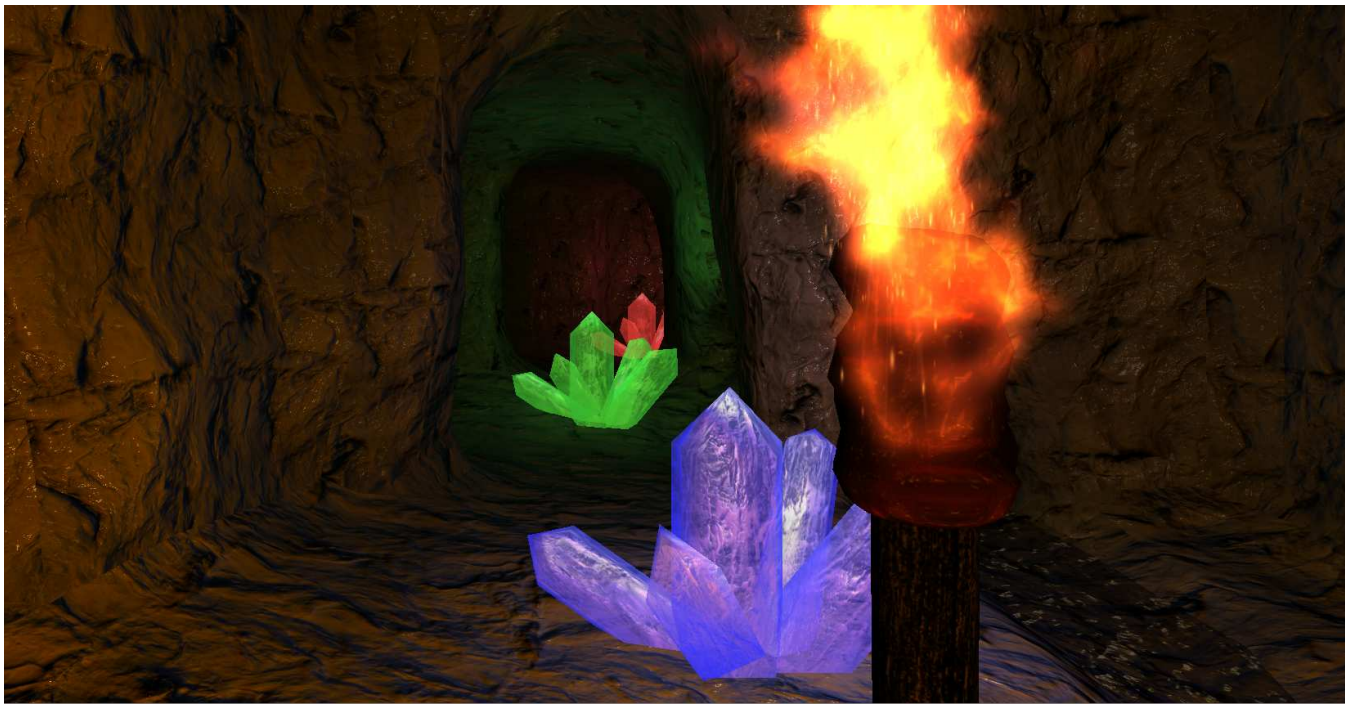

Figure 1: *Example project demonstrating the capabilities of bRenderer in an interactive graphics demo.*

Abstract

In this article, we present bRenderer, a basic educational 3D rendering framework *that has resulted from four years of experience in teaching an introductory-level computer graphics course at the University of Zurich. Our renderer is based on the observation that teaching a single basic but comprehensive computer graphics course often means to face the choice between students learning a low-level graphics API bottom-up on one side, or a powerful (game) engine on the other. Solutions between these two extremes tend to be either too rudimentary to easily allow advanced visual effects in student projects, or too abstract to facilitate learning about the underlying principles of computer graphics. Our platform-independent framework abstracts the functionality of its underlying graphics API and libraries to an extent that still preserves the main concepts taught in a computer graphics course. Consequently, bRenderer can be used in student projects, as well as in exercises. It helps students to easily understand how a renderer is implemented without getting distracted by the particular implementation of the framework or platform-specific characteristics.*

Categories and Subject Descriptors (according to ACM CCS): K.3.2 [Computers and Education]: Computer and Information Science Education—Computer science education

1. Introduction

Computer graphics (CG) tends to be a demanding subject for students, especially when taught at an undergraduate level, as it requires not only basic knowledge of linear algebra but usually also C++ programming, algorithms and solid software engineering skills. The situation has been further complicated by ongoing changes and updates in graphics hard- and software that in turn also changed the computer graphics curriculum at many universities. The University of Zurich has offered both theoretical (CG Lecture) and practical (CG Lab) computer graphics courses at undergraduate and graduate level for a number of years. The CG Lab course (6 ETCS points) depends on the simultaneous or prior completion of the CG Lecture (3 ECTS points) and is based on 6 weekly programming assignments followed by a final 6-week student project that encourages students to explore more advanced CG topics. For the practical CG Lab course we originally relied on classic immediate mode OpenGL examples, using the GLUT library [GLU] with a traditional fixed-function rendering pipeline. To follow the advancing CG standards, in 2013 we abandoned this approach in favor of OpenGL ES and iOS on the iPad platform, which we now use for exercises and student projects. This environment is widespread, well supported and allows for motivating projects despite the limitations of a single course, in terms of 3D content and CG functionality. Since then, we have also offered course participants a simple CG framework for programming the iPad, which is provided for the duration of the course. In 2014, we further updated the lab course to fully incorporate shaders into the framework, making better use of the iPad platform's freely programmable GPU.

When first exposing students to shader programming in a basic but still comprehensive CG course, we noticed that using shaders for more advanced features seemed to be difficult for many course participants, as few made extensive use of them. Although our framework simplified texture, geometry, and shader usage, students noted that features such as render-to-texture still required much OpenGL boilerplate code and knowledge of the internal framework implementation. Hence, although producing interesting results (see Section 3), course participants typically did not implement more advanced techniques like shadow mapping in their student projects, due to time constraints. We alleviated this issue by constantly improving the framework based on student feedback. In particular, we made the API easier to use and found more sensible abstractions for underlying graphics primitives (as explained in Section 4), still allowing full access to the underlying OpenGL ES implementation.

Our experiences illustrate the tradeoff that has to be considered when selecting or designing a framework for CG education. Finding the right software tools for such a course is not an easy task, as professional game engines may be too abstract and comprehensive to teach fundamental CG concepts, while large middleware, such as OGRE [ogr], might be difficult to understand for students who take their first steps in the field. Directly using a graphics Application Programming Interface (API) such as OpenGL, on the other hand, can be especially inefficient for student projects, since such an approach often requires great diligence of course participants to achieve even simple graphical effects.

Consequently, the last iteration of our framework, the *Basic Renderer* (bRenderer), has been considerably improved to aid teaching of topics such as geometric objects, transformations, illumination, shading, texturing and shadows, while at the same time effectively supporting students in implementing more advanced CG techniques. To this aim, we also provide an extensive online documentation of the framework. Note that our observations, experiences, and conclusions primarily apply to the scenario where CG and its practical application is taught in one single-semester course, thus with limited time available to cover all the fundamental concepts.

2. Related Work

As basis for a computer graphics course, it appears to be a straightforward solution to use a framework that has originally been intended for computer games development. Such an approach was chosen at ETH Zurich, where instead of a proprietary framework, MonoGame [Mon] is used in their Game Programming Laboratory [Gam]. The course is intended for master students who are already familiar with the basics of computer graphics, and focuses more on the development of a video game than on teaching elementary CG concepts. Structurally and in terms of features, MonoGame is similar to our framework. However, written in C# and based on Microsoft's XNA [XNA], MonoGame is a slightly more high-level solution that is nevertheless well-suited for a more advanced course with a focus on game development.

Conversely, driven by a trend to activity-led learning [AP09] and practical courses, various tools and libraries have been created and successfully used to teach computer graphics at an introductory level. Since these are often specifically tailored to the needs of the individual course or faculty at the university of origin, their priorities often vary widely. This is reflected in the implemented functionality, the selected programming language and middleware, as well as the architecture and design. Two noteworthy examples, that we briefly contrast with our bRenderer framework, are glGA [PPGT14] from the University of Crete and FUSEE [MG14] from the University of Furtwangen.

2.1. glGA

glGA is a simple cross-platform framework that adopts modern shader-based OpenGL and is written in C++. A major goal of the framework is to hide non-graphics related functionality and expose the GPU programming tasks, thus avoiding the steep learning curve of a more complete graphics engine such as OGRE.

Compared to bRenderer, glGA is a more minimalist framework that does not wrap its integrated libraries such as GLFW [GLF], GLEW [GLE] or Assimp [ass]. Instead, utility classes are provided to alleviate basic tasks, as for instance the loading of meshes, and only wrap a small portion of the functionality provided by the libraries. Students therefore have to familiarize themselves with a variety of different APIs instead of just one, as in our approach.

2.2. FUSEE

The Furtwangen University Simulation and Entertainment Engine (FUSEE) is a cross-platform framework written in C#. Using the Xamarin cross-compiler [Xam], it is possible to export applications

to native Android and iOS, and even HTML 5, via translating the framework's core and the user-written application from the original C# bytecode into JavaScript.

FUSEE further offers the possibility to either render an object directly, or to add it to a scene that can be traversed by a *Scene Manager* later on. This class provides two features that are separately implemented in our renderer as *Object Manager* and *Render Queue* (see Section 4.5). Another feature, offered by both FUSEE and bRenderer, is the possibility to either use simple shaders provided by the framework, or to write custom shaders for more advanced effects. Whereas FUSEE offers only a few prefab shaders, our bRenderer framework is also able to generate application-specific shaders, according to the user's requirements (see Section 4.6).

Additionally, FUSEE features audio and physics support; this is currently not offered by our solution, which is designed as a rendering framework only.

3. Student Projects and Exercises

During our CG Lab course, students have to first successfully complete 6 exercises (one exercise per week) covering elementary topics, such as the basics of shader programming, the calculation of face and vertex normals, (hierarchical) transformations, lighting and shading, the usage of textures, as well as more advanced topics, such as the implementation of normal mapping. Representative for such an exercise is the following task:

> Program a crude animated model of the solar system, including the sun, all planets, and several moons, using hierarchical transformations (by applying the appropriate transformation matrices in the right order).

As a starting point for the respective exercise, students are provided with an example project that just draws a sphere using our framework. They are then required to extend the given program by creating and applying transformation matrices, writing corresponding vertex and fragment shaders, etc. For each exercise, students can collect a certain number of points, based on what tasks they successfully complete and how well they manage to do so. In the first half of the course, students are required to collect a minimum number of points to be allowed to continue.

In the second part of the course, provided that they successfully completed the exercises, students are required to complete a project work in small groups of about 3 people. In previous years, usually about 19 students participated in 7-8 projects during the course. Within 6 weeks, students have to propose, design, plan, and implement a computer game project that is eventually to be presented in class. As they can only pass or fail in our course, this decision depends on how well students accomplished the initially proposed goals for their projects, also in comparison with their peers. As part of these goals, students are also required to implement more challenging techniques; recent examples are shadow mapping or post-processing effects. These can be proposed by the students themselves, or are suggested by the teacher of the course for each project individually during the project's early design phase.

Examples of student projects from recent years are depicted in Figure 2. The example from 2016 makes use of shadow mapping,

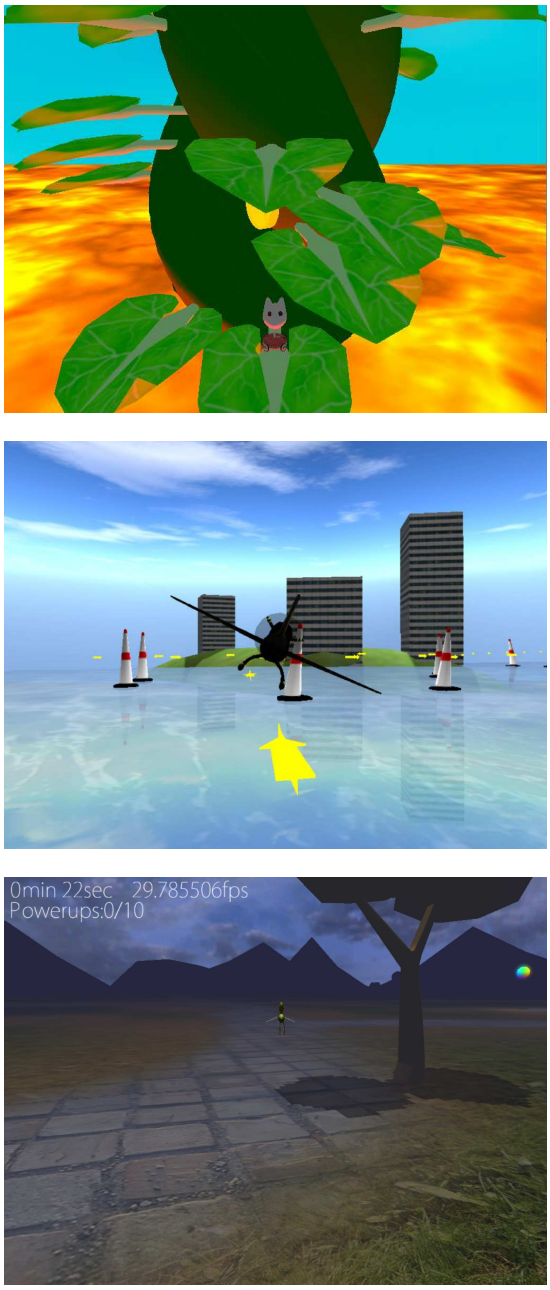

Figure 2: *Screen-shots from student projects of recent years, 2014 to 2016 (top to bottom).*

more advanced shaders for rendering ground and trees, distance fog, as well as a bloom post-processing effect. These effects are more clearly visible in a captured video of this game, available on the bRenderer homepage, see Section 4. The earlier projects, on the other hand, typically used comparably simple techniques, such as very straightforward cartoon-style shading (2014 example), or relatively easy to achieve effects like reflective water surfaces based on environment mapping (2015 example). Similar to other more simplistic frameworks, such as glGA, previous iterations of our frame-

work required students to write lots of error-prone OpenGL boilerplate code to achieve effects, such as those used by aforementioned student project of 2016. In contrast, by providing sensible abstractions for OpenGL framebuffer objects, as well as render-to-texture and post-processing effects, our recent framework makes related techniques much easier to implement. In fact, over the previous years, we observed that the techniques students were able to implement in their projects became more complex as we improved the abstractions that our framework provides, leading to its current iteration, bRenderer. As the most recent project showcases, our framework allows students to engage with more complex CG topics much easier than previously feasible. Additional features, such as text rendering, as shown by the same example in Figure 2, further alleviate the development of computer games.

4. Framework Description

bRenderer was developed as a cross-platform 3D rendering framework that is specifically tailored to the needs of a computer graphics course. It is based on the experiences of several years of teaching computer graphics, as well as the code base of the previous iteration of our framework. bRenderer abstracts the functionality of its underlying graphics API and libraries to an extent that still allows students to easily grasp the concepts and techniques to be taught.

Compared to glGA or our previous framework, which provide abstractions for only the most essential elements required to draw models, such as shaders, meshes and textures, bRenderer offers a large number of features with a consistent interface (see Section 4.1). Simultaneously, it remains close to the native underlying graphics API and the C++ programming language, unlike FUSEE for example.

Instead of writing complex OpenGL function calls, more advanced techniques such as drawing to texture, can be achieved with only a few lines of code, as illustrated in the following listing. The example code shows the creation of a framebuffer and a texture object. In the render loop, the framebuffer can be bound and a texture to draw to may be specified. A Boolean parameter determines whether the previously active framebuffer will be bound again, when unbinding the custom framebuffer object.

```
1 FramebufferPtr fbo = bRenderer().getObjects()
      ->createFramebuffer("fbo");
2 TexturePtr fbo_texture = bRenderer().
      getObjects()->createTexture("fbo_texture",
       0.f, 0.f);
3 ...
4 /* in render loop */
5 fbo->bindTexture(fbo_texture, true);
```

Our framework's well-documented code base is accompanied by online guides and a comprehensive example project demonstrating the renderer's capabilities, as shown in Figure 1. This allows students to quickly familiarize themselves with the API and easily comprehend how to achieve also advanced effects, or how to extend and customize the framework.

Both the documentation and the full source code including external libraries and the example project are provided at the bRenderer homepage that can be accessed via http://ifi.uzh.ch/vmml.

Our framework is deliberately written in C++ to the largest extent feasible, as the programming language is very prevalent in CG development, unlike the iPad platform's native ObjectiveC language, or more abstract languages like C#, as used by the FUSEE framework. Moreover, many students already have some experience in C++ from other courses. As a rendering module, bRenderer can be integrated into applications very easily by inheriting an abstract class called IRenderProject, as shown in Figure 3. The application is then required to implement only three callback functions in order to ensure minimal compatibility with the renderer:

- An *initialization function* that is called during the setup of the renderer, responsible for the loading of models, creation of lights and cameras, the definition of variables, etc.
- An *update function* that is called every frame for updating and drawing the scene or loading and creating new objects if required.
- A *finish function* is called when the renderer is shut down and allows the deallocation of memory and other resources.

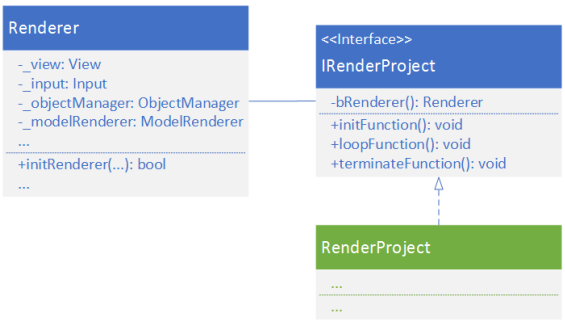

Figure 3: *A project using the renderer inherits the abstract class IRenderProject.*

A project inheriting the aforementioned abstract class automatically holds an instance of the renderer, which itself maintains a pointer to the project for callbacks. In the application, the renderer can then be simply acquired via an accessor function.

4.1. Feature Overview

bRenderer is equipped with a variety of tools to enable beginners to easily draw geometry to the screen and facilitate creating advanced effects, later in a computer graphics course. To ensure a gradual learning curve, many tasks may be completed at different levels of abstraction and complexity.

Students new to computer graphics are able to load 3D models with a simple function call to the object manager, one of the four main modules of the renderer (explained in Section 4.2), only passing the filename as an argument. The material is then loaded automatically, and suitable shaders are generated. In the loop function of an application, another method provided by the model renderer (see Section 4.2) can be invoked to draw a model. This method

simply takes the model itself, the model matrix, the view matrix and the projection matrix as arguments. As a result, the instructor of the course is able to discuss the mathematical principles of CG without initially overwhelming the class with shader programs and implementation details of the framework.

As soon as the students are familiarized with the framework, objects such as cameras and lights may be introduced and custom shader files can be loaded to accomplish more sophisticated effects. To pass supplementary shader uniform variables, an instance of the renderer's *Property* class can be created. Alternatively, uniform variables can be passed directly to the shader.

Experienced programmers benefit from the customizability and extensibility of the renderer and may invoke functions using many more arguments, which are otherwise defined by default values.

To achieve post-processing effects and to enable multi-pass rendering, regular textures or even cube- and depth-maps can be bound to the framebuffer object abstraction provided by the framework. The concepts of how to create shadows and reflections in a 3D scene can thus be learned without the numerous function calls to OpenGL, which have proven to be difficult to use for students in a compact graphics lab.

Another feature worth to be noted is the support of text rendering. Fonts are loaded with the aid of the FreeType library [frec] and can be drawn as text sprites. The texture atlas containing the characters may be applied to custom models as well, since it can be obtained from the framework's Font class via a simple accessor function.

The following list presents the most important features of our renderer within the limited scope of this report:

- Cross-platform compatibility
- User inputs
- Windows (and views on iOS) with OpenGL context
- Object management
- Adjustable global configurations
- Queuing render calls (transparency sorting and sorting for efficiency)
- Bounding volumes and culling
- OBJ loading (materials and models)
- Sprites and text sprites
- Fonts
- Textures (including cube maps and depth maps)
- Framebuffers (allow drawing to textures)
- Shaders
  - Loading shaders from files
  - Generation of shaders according to user specifications or material data
- Camera objects
- Light objects

4.2. Design

We designed our renderer with the Model-View-Controller (MVC) pattern in mind. Its functionality is therefore divided into modules as illustrated in Figure 4.

The information needed to draw geometry to the screen is stored as objects such as models, cameras and lights. Those are maintained in the *ObjectManager* class (model in MVC), allowing objects to be loaded, created, accessed and removed at any time without the risk of memory leaks, since every object name can exist only once and reference counting provided by smart pointers removes objects as soon as they are no longer required.

The framework maintains a view on iOS or a window on desktop systems that displays the rendered images. To provide a coherent interface on all platforms, this functionality is abstracted by the *View* class. It offers a multitude of options to adjust it to the needs of an application, including changing its position or resolution and delivers useful information such as the current aspect ratio or the resolution of the screen being used. Objects stored in the object manager may be drawn to the view either directly, or they can be queued and sorted first using the renderer's model renderer.

Although the user's project controls what is to be displayed on the screen, the renderer provides a useful tool that allows for the user to interact with the application. This happens in the form of the *Input* class, which is closely coupled with the view. The abstraction of user input is performed on a low level, as touch events, for example. Consequently, high level abstractions, such as object picking, must be provided by the application, if required.

The modular design of our framework allows to adjust it to the application's needs as the modules concerning view and input do not rely on the modules managing and rendering the objects.

To simplify using the renderer, we created the *Renderer* class. It is meant as the main gateway to the framework through which most functionality can be setup and accessed. It maintains one instance of each of the four building blocks of the framework (as illustrated in Figure 4) to be able to draw objects to the screen. The Renderer class allows for initializing its modules and running the render loop with only two lines of code.

In the example code below, the renderer is initialized with optional parameters to set the resolution, full screen mode, and title. To achieve a similar initialization in glGA (Section 2.1), significantly more complex code needs to be written by the user, including various calls to external libraries and OpenGL.

```
1 /* initialize with optional parameters */
2 bRenderer().initRenderer(1920, 1080, false,
3   "The Cave - Demo");
4 /* start main loop */
5 bRenderer().runRenderer();
```

4.3. Cross-Platform Development

bRenderer was designed as a cross-platform framework running on Windows, Mac, Linux and iOS and provides a consistent API on all platforms. Our platform-specific windows and views are abstracted by the View class and the closely coupled Input class. Whereas the former provides an OpenGL context to present the rendered images on the screen, the latter allows access to user inputs concerning the view. Depending on the platform, inputs may originate from mouse and keyboard or a touchscreen, respectively.

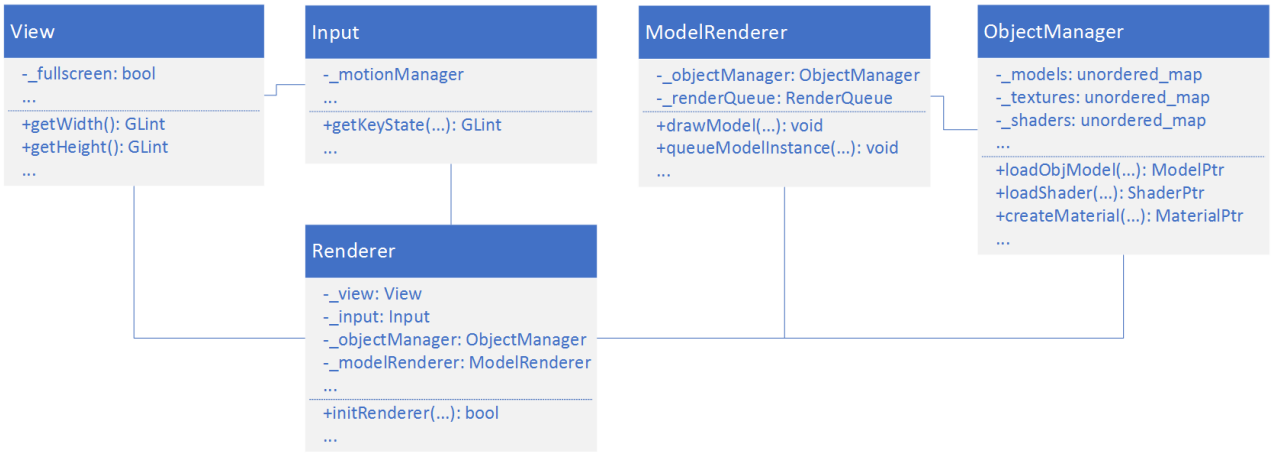

Figure 4: *The framework's functionality is provided by four major modules.*

Our framework automatically recognizes the operating system (OS) it is being compiled for and defines preprocessor macros in a header file. It internally decides which implementations of the View and Input classes are to be used, and for its objects relying on OpenGL, compatible calls to the graphics API are chosen. Thus, the renderer can be used on all supported platforms in the same manner and code written for a specific platform will not cause errors when an application is ported to another supported OS.

4.4. Libraries

The choice of the underlying libraries is based on the idea of keeping the framework lightweight, modern and easily comprehensible. For this reason we did not consider GLUT [GLU] for our renderer as it is not actively supported anymore, nor is it open source. FreeGLUT [Frea] as an open source alternative is still actively maintained, yet based on the deprecated GLUT API.

We therefore chose GLFW as a small and modern C library to complement our renderer on desktop systems. During the course of development, we experienced it as reasonable and easily understandable. Since GLFW does not support iOS, we used the UIKit [UIK] framework supplied by Apple to implement our own view for the iPad platform.

Additionally, we added GLEW [GLE] for loading OpenGL extensions, FreeType [frec] and Freetype-gl [fred] to support for a wide variety of font formats, and FreeImage [freb] to enable image support for texture creation.

Since the underlying geometric concepts of 3D models are a major part of our curriculum, we tried to keep them accessible by choosing the human-readable Wavefront OBJ format and parse the files with libobj [lib].

With the exception of the math library vmmlib [vmm], libraries are abstracted by the classes that use them and are not supposed to be used directly, thus ensuring a simple and consistent API across different platforms.

4.5. Drawing

Drawing in our renderer is based on objects implementing the IDrawable interface, having the single responsibility to draw the data they hold. This data is often originally provided by an accompanying data class. Additional classes can be used to define certain properties of the object to be drawn, including camera and light classes, as well as shaders and textures.

To simplify the drawing process for unexperienced programmers, we created the model renderer. It is one of the four modules introduced in Section 4.2 and allows for drawing objects with a simple function call. The model renderer is capable of drawing to the view directly or queuing the draw call using a render queue.

The render queue allows for storing and caching draw calls. Its main feature is the ability to sort draw calls according to distance (for transparency sorting) or by material properties (such as shaders and textures) to avoid costly state changes as much as possible.

4.6. Shaders

Our renderer is based on the programmable graphics pipeline and thus relies on shaders to draw its objects. To ensure compatibility with older systems, the shader and corresponding data classes cover only features that are part of OpenGL ES 2.0. Among others, geometry shaders or uniform buffer objects are therefore not yet supported by our renderer.

bRenderer allows experienced users to load their shaders from custom shader files and additionally offers a shader generator for quick and easy experimenting. Thus, our renderer features a collection of building blocks to create a large variety of shaders according to the requirements, given by either material data or custom settings.

5. Example Project

Over the course of development, the implemented features of our renderer have constantly been tested and improved. To allow for

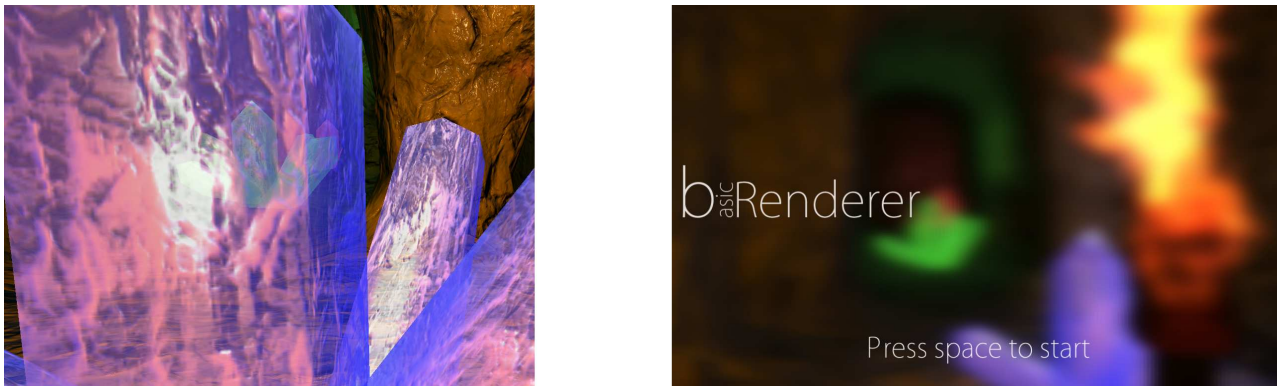

Figure 5: *The example project demonstrates transparency sorting and post-processing effects.*

assessing the techniques and their impact on performance, we created an example scene. It should be visually appealing to increase the motivation to work with the framework and illustrate most of its capabilities, as well as its functional principles. It may therefore act as a guide for programmers to gain a first impression on how the provided functionality can be used, and might serve as a starting point for their own applications.

The project was designed as an interactive graphics demo that enables the user to explore a cave sparsely illuminated by glowing crystals. To demonstrate the dynamic lighting capabilities of the framework, we added a torch to the scene that moves with the viewer to illuminate the environment close to the camera. An example result is shown in Figure 1 and was successfully tested on Windows, Linux, Mac OS X, and iOS.

On desktop platforms, the camera can be controlled using the mouse to look around, as well as with the typical WASD key scheme for movement. It additionally may be tilted and shifted up and down with the arrow keys, so all possibilities for rotating and moving the camera are demonstrated. On iOS, the left half of the touchscreen is treated as the left stick of a game controller and is responsible for movement, the right half is used to look around. The touchscreen controls are illustrated in Figure 6. It is thus possible to freely fly through the scene on all platforms, to closely inspect the visual results produced by the different shaders applied to the models.

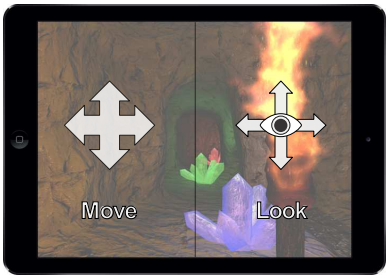

Figure 6: *The movement and camera controls on an iPad screen.*

Since the renderer often provides multiple possibilities to achieve a certain goal, the example project shows different strategies for loading, creating and drawing a multitude of objects, which are further explained in the form of comments. The following example code demonstrates how objects are constructed in the initialization function. Some shaders and materials are created separately, whereas others are automatically read in or generated when loading an OBJ model or building a sprite.

```
1  /* load materials and shaders */
2  MaterialPtr flameMaterial = bRenderer().
       getObjects()->loadObjMaterial(
3    "flame.mtl", "flame", flameShader);
4  ...
5  /* create properties for a model */
6  PropertiesPtr flameProperties = bRenderer().
       getObjects()->createProperties(
7    "flameProperties");
8  ...
9  /* load models */
10 bRenderer().getObjects()->loadObjModel_o(
11   "crystal.obj");
12 bRenderer().getObjects()->loadObjModel_o(
13   "cave.obj", 4, FLIP_T | FLIP_Z |
         VARIABLE_NUMBER_OF_LIGHTS);
14 /* create sprites */
15 bRenderer().getObjects()->createSprite(
16   "sparks", "sparks.png");
17 bRenderer().getObjects()->createSprite_o(
18   "flame", flameMaterial, NO_OPTION,
         flameProperties);
```

The example first illustrates how a custom material can be loaded from a material file (line 2). A custom shader can be passed optionally, alternatively the renderer generates a shader automatically. Additional properties to be passed to the shader are created in the next step (line 6).

When a model is loaded, we pass the name of the file to the function as the only mandatory argument (line 10). The example code shows how the model of a cave is loaded with additional parameters to automatically create a shader with a maximum number of four light sources, which is variable (line 12). This means that a lower number of lights might be passed to the shader when drawing the model. In addition, the T-axis of the texture coordinates and the Z-axis of the model are flipped.

As shown at the end of the code example, a sprite can be created by passing only two arguments (its name and a texture, as in line 15). Still, our framework allows for a lot of customizability and flexibility by allowing to pass custom materials and additional properties (line 17). This principle applies to all objects and is an important part of the framework's design that eases the first steps for students at the beginning of a CG course, yet allows them to achieve advanced effects at a later stage.

In contrast, previous iterations of our framework did not offer the option to generate shaders, reuse materials and shaders for multiple models, or pass flags to customize the creation and loading of objects. Thus, our framework has gained flexibility and customizability while it remained simple to use, as such customizations are completely optional.

The main part of the scene is drawn using the render queue (Section 4.5), allowing the crystals to be partially translucent due to alpha blending. At startup and when pausing the demo, the queue is not directly displayed, as the scene is blurred in a post-processing step and a logo is shown (Figure 5). The 3 sprites necessary for this effect are drawn directly to the screen, demonstrating the immediate drawing of models as well.

As the renderer is supposed to be used in computer graphics courses, other post-processing effects such as reflections were removed from the example project after testing was successful. This allows for using the example project as a guide for students, without giving away solutions for the assignments.

6. Discussion

Computer graphics is a demanding field for students and software development alike. Beginners are confronted not only with linear algebra and geometric concepts but also with geometric modeling, color science, lighting and shading and a multitude of libraries and APIs. To achieve a broad understanding of the underlying concepts may therefore take up a lot of time and requires innovative learning strategies. Those can be based on both theoretical lectures and practical courses, accompanied by accessible, yet powerful software tools and frameworks.

We developed bRenderer as a cross-platform 3D rendering framework, which is specifically tailored to the needs of a modern, practical and compact computer graphics course. It abstracts the functionality of its underlying graphics API and libraries to an extent that does not conceal the concepts to be taught, without being limited in functionality. The well-documented code base is accompanied by online guides and a comprehensive example project demonstrating the renderer's capabilities. This allows to quickly familiarize with the API and to easily comprehend how to achieve advanced effects or even extend and customize the framework.

However, we designed bRenderer as a rendering framework only and it is therefore limited in scope. Whereas purely graphical tasks may be performed quickly using the renderer, development of game mechanics, animations or physics simulations is not covered by its implementation. However, it is easily possible for students to access bounding volumes and vertices of geometric objects to extend the framework with such functionality.

Another limitation may arise from the choice to use the same OpenGL features and utilizing the same shader programs on all platforms. While this reduces the complexity of the code base and facilitates platform-independence, it currently constrains the features that can be used to the ones offered by OpenGL ES 2.0.

On the other hand, the basic design of our renderer offers flexibility and modularity, allowing to easily integrate it into existing projects. Moreover, its customizable and extensible nature as well as its comprehensive feature list qualifies our framework not only for taking the first steps in computer graphics but also for the use in more complex applications and games.

References

[AP09] ANDERSON E. F., PETERS C. E.: On the Provision of a Comprehensive Computer Graphics Education in the Context of Computer Games: An Activity-Led Instruction Approach. In *Eurographics 2009 - Education Papers* (2009), Domik G., Scateni R., (Eds.), The Eurographics Association. doi:10.2312/eged.20091012. 2

[ass] Assimp. http://www.assimp.org/. Accessed: 12.01.2016. 2

[Frea] Freeglut. http://freeglut.sourceforge.net/. Accessed: 03.09.2015. 6

[freb] Freeimage. http://freeimage.sourceforge.net/. Accessed: 08.09.2015. 6

[frec] Freetype. http://www.freetype.org/. Accessed: 31.08.2015. 5, 6

[fred] freetype-gl. https://code.google.com/p/freetype-gl/. Accessed: 08.09.2015. 6

[Gam] Game programming laboratory. https://graphics.ethz.ch/teaching/gamelab16/home.php. Accessed: 19.02.2017. 2

[GLE] Glew. http://glew.sourceforge.net/. Accessed: 30.08.2015. 2, 6

[GLF] Glfw. http://www.glfw.org/. Accessed: 30.08.2015. 2

[GLU] Glut. https://www.opengl.org/resources/libraries/glut/. Accessed: 03.09.2015. 2, 6

[lib] libobj. http://people.cs.kuleuven.be/~ares.lagae/libobj/. Accessed: 05.09.2015. 6

[MG14] MÜELLER C., GÄRTNER F.: Student Project - Portable Real-Time 3D Engine. In *Eurographics 2014 - Education Papers* (2014), Bourdin J.-J., Jorge J., Anderson E., (Eds.), The Eurographics Association. doi:10.2312/eged.20141030. 2

[Mon] Monogame. http://www.monogame.net/. Accessed: 19.02.2017. 2

[ogr] Ogre. http://www.ogre3d.org/. Accessed: 12.01.2016. 2

[PPGT14] PAPAGIANNAKIS G., PAPANIKOLAOU P., GREASSIDOU E., TRAHANIAS P.: glGA: an opengl geometric application framework for a modern, shader-based computer graphics curriculum. In *Eurographics 2014 - Education Papers* (2014), Bourdin J.-J., Jorge J., Anderson E., (Eds.), The Eurographics Association. doi:10.2312/eged.20141026. 2

[UIK] Uikit. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/. Accessed: 30.08.2015. 6

[vmm] vmmlib. https://github.com/VMML/vmmlib. Accessed: 30.08.2015. 6

[Xam] Xamarin. https://xamarin.com/. Accessed: 13.09.2015. 2

[XNA] Xna. https://www.microsoft.com/en-us/download/details.aspx?id=23714. Accessed: 19.02.2017. 2